ECOLE
POLYTECHNIQUE
DE BRUXELLES

M-IRIFS:1 — 2020-2021

Heuristic Optimization

Implementation exercise 2

Author:
Maxime LANGLET

Professor:
Dr. T. Stüzle

3rd of May, 2021

Contents

1 Introduction

During the course INFO-H413 - Heuristic optimization, it is asked to implement two stochastic local search algorithms for the permutation flow-shop problem (PFSP) with the sum weighted completion time objective (PFSP-WCT) building on top of the constructive and perturbative local search methods from the first implementation exercise. In particular, in this report, the two algorithm implemented are the Iterated Greedy Algorithm (IGA) [1] and the Tabu Search procedure [3]. These algorithms were chosen since they belong to different classes, Tabu Search being a simple SLS method and IGA being a particular case of an ILS, and thus an hybrid SLS method
The algorithms were implemented in C++, and were run sequentially using Python, the testing was done using R. These algorithms were tested on the same collection of instances from implementation exercise 1.

1.1 Running the code

In order to run the program on a particular instance, this command line will run the executable :

```
$ make && ./flowshopWCT <instance_file> <IGA/Tabu> <random/srz>
```

Note that one needs to be in the flowshopWCT folder in order to successfully run this command. Note also that the instance file needs to be the relative path to that folder, for example ../path/to/instance/instance_file.txt. In the testing, the use of a Python script was done to run successively all instances of the same size. Some parameters needs to be modified from one test to another but these are detailed in the file **make.py**. To run it, the next command will do :

```
$ python src/make.py
```

2 Algorithms presentation

In this section, we will present the algorithms implemented by means of a pseudo code for each one respectively. Note that the cost function is noted as $F()$ and computed the WCT for a given solution. The stopping criterion is 500 times the average computed time that it takes to run a full VND in the preceding implementation exercises. In our case, this yields a stopping time of $22s$ for instances with 50 jobs and $333s$ for instances of 100 jobs. In both cases, the generation of the initial solution is based on the results of the first implementation exercises. Thus, the simplified RZ method is used in the implementation.

2.1 Iterative Greedy Improvement Algorithm

The IGA is pretty straight forward, it starts from an initial solution and iterates over a main loop which has two phases, one destructing the initial solution and another reconstructing it. The pseudo code is the following :

Algorithm 1: Iterative Greedy Improvement Algorithm

```
Result: New candidate solution
π_0 ← GenerateInitialSolution();
π ← LocalSearch(π_0);
while termination criterion is not met do
    π' ← Destruction_Construction(π)      % ran 10 times, taking only the best perturbation;
    π'' ← LocalSearch(π');
    if F(π'') < F(π) then
        π ← π'';
return π
```

In the article [1], the *LocalSearch()* method is based on an insertion neighborhood, which also the case in our implementation. The insertion method is thus the same from the preceding implementation exercises. Let's present the pseudo code of the *Destruction_Construction()* method:

Algorithm 2: Destruction_Construction

Data: π a candidate solution, d an integer
Result: Best permutation after insertion of randomly selected d jobs
Set π^R empty;
$\pi' \leftarrow \pi$;
for $i \leftarrow 1$ *to* d **do**
  $\pi' \leftarrow$ remove a randomly selected job from π';
  $\pi^R \leftarrow$ include the removed job into π^R;
for $j \leftarrow 1$ *to* d **do**
  $\pi' \leftarrow$ best permutation obtained after inserting job π_j^R in all possible positions of π';
return π'

The destruction phase removes randomly d jobs from the current solution. It then follows the construction procedure which applies a greedy constructive algorithm to reconstruct a solution by reinserting those jobs. This function is called multiple times in a row on the current solution(10 times to be exact), and taking only the best disturbed solution. Furthermore, the d parameter was tuned to get the best average relative percentage deviation. Surprisingly, the tuning of this parameter was found to be the same as the tuning from the paper [1] (i.e. $d = 8$), although only 3 parameters were tested ($d \in \{6, 8, 10\}$) since the computational time was quite lengthy. It would be interesting to test more parameters than the ones presented in this report.

2.2 Tabu Search procedure

The Tabu Search procedure uses a tabu list to remember the solutions that had been recently examined to avoid repetition. The neighborhood structures are used in this algorithm. A neighborhood N of a solution π can be found by applying the insertion operator on this solution. The insertion operator was used because we concluded from the preceding implementation exercise that it performed better overall. The insertion of an element is limited the parameter α, which defines the extent of the insertion. Let's say that $\alpha = 3$, then the element can be inserted to one of the three locations before or after it's original location, this method will be called $NSP()$, which stands for neighborhood search procedure. Furthermore, the *perturbation()* method is in fact a toned down *Destruction_Construction()* method. It present the same algorithm than before, but rather than picking a large number of jobs, we only take a few. For example, the IGA algorithm had a parameter d of 8, now we have $d = 2$. We also run the perturbation algorithm multiple times per given solution like to the preceding method, giving on average better quality solutions. This was thought acceptable since it induces a slight perturbation on the given candidate. The initial pseudo code is the

following :

Algorithm 3: Tabu Search method

```
Data: π an initial solution
Result: π* a new improved candidate solution
π* ← π ;
c* ← F(π) ;
stuck ← 0 ;
while stopping criterion is not met do
    bestMove ← NSP(π, α);
    addToTabuList(bestMove)        % implemented in the NSP method but noted here;
    c ← F(bestMove) ;
    if c < c* then
        c* ← c;
        stuck← 0;
    else
        stuck←stuck+1;
        if stuck ≥maxStuck then
            perturbation();
            stuck←0;
return π*
```

Originally, the stopping criterion is denoted by a maximum number of iterations, but in our case, this criterion is timing based. In the paper [3], 2 numbers of *maxStuck* are proposed and it was decided to take 10 as the *maxStuck* criterion. Let's detail the *NSP()* method. It might differ slightly from the original method presented in the article [3]. In particular, the method always return the best configuration given a candidate solution and the parameter α. So it might happen that the returned solution is the initial solution, since a condition over the cost function is applied right after computing this cost. Here is the pseudo code:

Algorithm 4: NSP neighborhood search procedure

```
Data: π an initial solution, α a parameter
Result: π* a new improved candidate solution
π* ← π ;
c* ← F(π) ;
for j ← 1 to π.size() do
    for i ← j − α to j + α do
        π ← Insert(j, i)                % inserting element j to position i;
        c ← F(π*);
        if c < c* and π* not in Tabu_List then
            c* ← c;
            π* ← π;
            Tabu_List add π;
return π*
```

Of course, it is implied that the indices i and j respect the size constraints of the Tabu_list.

3 Results

As noted before, the stopping criterion for all instances was 500 times the average computing time to run a full VND. This defines the stopping criterion at 22 seconds for small instances and 333 seconds for bigger instances. Each algorithms were run 5 times on each instances, the following results will be an average over those 5 runs. Some files can be found in the *instances* folder presenting those averages, there are called

*IGA/Tabu_*50/100.*txt* and the averages were casted as integers. It is then asked to compute the average relative percentage deviation from the best solution known on every instances. Reminder, the relative percentage deviation (RPD) is given by the formula below.

$$\Delta_{ki} = 100 * \frac{cost_{ki} - best\_known_i}{best\_known_i}$$

Before presenting the averages, a R notebook was used to derive those results, this file being in the *instances* folder and named *R_test.ipynb*. The following table summarizes the average percentage deviation for each size of instances :

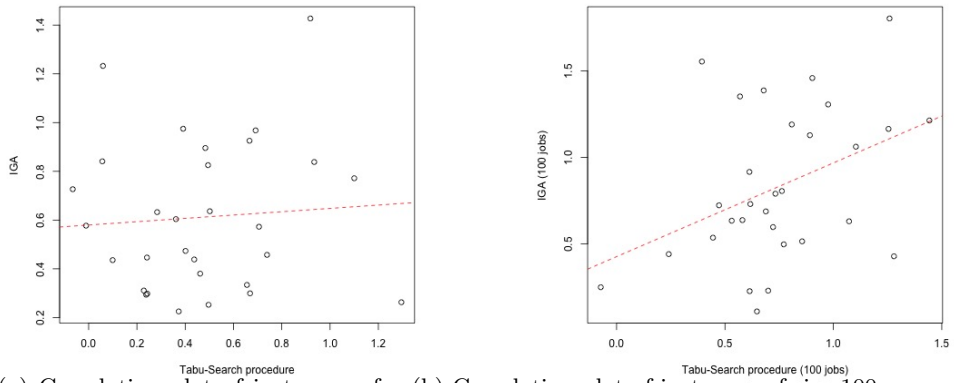
Algo\Instances size	50	100
IGA	0.472	0.752
Tabu	0.612	0.833

Table 1: Average percentage deviation per instance size

We can see that, on average, the IGA performs slightly better than the Tabu Search, although the solution quality isn't far from one another. Recall that the best performing configuration of the VND algorithm gave on average a RPD of 2.202 in the preceding implementation exercises. It is hard to say which one yields the better average solution since our implementations run 500 times longer.

Since this comparison is outside the scope of these exercises, an unrigorous approach was taken. The same algorithm from the last implementation exercise was used with the exception of the stopping criterion is now the same as the one in our current implementation. Since there are no perturbations in the VND algorithm or randomness involved (the initial solution was generated using SRZ), repeating the algorithm will give the same results. We are left with a RPD of 1.987 when running the VND algorithm on an instance of 50 jobs for 22 seconds. This is quite a significant gap compared to our two implemented algorithms. We can say that the implemented algorithms perform better than the VND method, although the methodology isn't perfect. For instance, it is entirely possible that the algorithm is stuck in a local minima, but as noted before, this is a little bit outside the scope of these exercises and thus this is not verified.

The question now being is there a correlation between both solutions generated between both implemented algorithms. The following figure present the correlation plots for both instance sizes.



(a) Correlation plot of instances of size 50

(b) Correlation plot of instances of size 100

Figure 1: Correlation plots for both instances size

Note that the red dotted lines were drawn using a linear regression on the scattered points. This was an attempt to model the relationship between both implementation. We can see that we have a relation

between both algorithm for bigger instances. The line drawn seem to indicate that IGA generates slightly better solutions on average (since the slope of the curve is lesser than 1), although when a good quality solution is difficult to generate, both algorithms have the same issues. It would be interesting to verify this trend with bigger instances. Furthermore, the instances of size 50 doesn't present the same behavior. The points present in the plot seem to be distributed more randomly. Thus the same conclusion can't be made for smaller instances.

Statistical tests

Here is the results of statistical tests :

	t-test	Wilcoxon
IGA and Tabu (50 jobs)	0.085	0.073
IGA and Tabu (100 jobs)	0.307	0.328

Table 2: Statistical tests

In both cases, the p-value being greater than the significance level ($\alpha > 0.05$), we cannot conclude that both algorithms are statically different from one another, although a p-value way larger for bigger instances is present, hence the behavior observed previously. This brings the question what causes the indifference between both implementations ? Firstly, both algorithms implement a local searching method using the *insert* procedure (*LocalSearch()* for IGA and *NSP()* for Tabu Search), this might create similar solutions, even if the neighborhood of the insert is different. Secondly, likewise with the local search method, the perturbation method are similar between both algorithms.

4 Conclusion

In conclusion, this second implementation exercise allowed us to learn more about improved stochastic local search algorithms for the permutation flow-shop problem. We also had the chance to tune and modify slightly those algorithms to fit our needs, giving us a larger view of the possibilities for future studies.

References

[1] Quan-Ke Pan and Rubén Ruiz. "Local search methods for the flowshop scheduling problem with flowtime minimization". In: *European Journal of Operational Research* 222.1 (2012), pp. 31–43. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2012.04.034. URL: https://www.sciencedirect.com/science/article/pii/S0377221712003426.

[2] Rubén Ruiz and Thomas Stützle. "A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem". In: *European Journal of Operational Research* 44 (Mar. 2007), pp. 2033–2049. DOI: 10.1016/j.ejor.2005.12.009.

[3] Lin-Yu Tseng and Ya-Tai Lin. "A genetic local search algorithm for minimizing total flowtime in the permutation flowshop scheduling problem". In: *International Journal of Production Economics* 127.1 (2010), pp. 121–128. ISSN: 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2010.05.003. URL: https://www.sciencedirect.com/science/article/pii/S0925527310001830.